Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Asynchronous I/O Support in Linux 2.5

Suparna Bhattacharya
Steven Pratt
Badari Pulavarty
Janet Morgan
IBM Linux Technology Center
suparna@in.ibm.com, slpratt@us.ibm.com,
pbadari@us.ibm.com, janetmor@us.ibm.com

Abstract

This paper describes the Asynchronous I/O (AIO) support in the Linux® 2.5 kernel, additional functionality available as patchsets, and plans for further improvements. More specifically, the following topics are treated in some depth:

- Asynchronous filesystem I/O
- Asynchronous direct I/O
- Asynchronous vector I/O

As of Linux 2.5, AIO falls into the common mainline path underlying all I/O operations, whether synchronous or asynchronous. The implications of this, and other significant ways in which the design for AIO in 2.5 differs from the patches that existed for 2.4, are explored as part of the discussion.

1 Introduction

All modern operating systems provide a variety of I/O capabilities, each characterized by their particular features and performance. One such capability is Asynchronous I/O, an important component of Enterprise Systems which allows applications to overlap processing with I/O operations for improved utilization of CPU and devices.

AIO can be used to improve application performance and connection management for web servers, proxy servers, databases, I/O intensive applications and various others.

Some of the capabilities and features provided by AIO are:

- The ability to submit multiple I/O requests with a single system call.
- The ability to submit an I/O request without waiting for its completion and to overlap the request with other processing.
- Optimization of disk activity by the kernel through combining or reordering the individual requests of a batched I/O.
- Better CPU utilization and system throughput by eliminating extra threads and reducing context switches.

2 Design principles

An AIO implementation can be characterized by the set of design principles on which it is based. This section examines AIO support in the Linux kernel in light of a few key aspects and design alternatives.

2.1 External interface design alternatives

There are at least two external interface design alternatives:

- A design that exposes essentially the same interfaces for synchronous and asynchronous operations with options to distinguish between mode of invocation [2].
- A design that defines a unique set of interfaces for asynchronous operations in support of AIO-specific requirements such as batch submission of different request types [4].

The AIO interface for Linux implements the second type of external interface design.

2.2 Internal design alternatives

There are several key features and possible approaches for the internal design of an AIO implementation:

- System design:
  - Implement the entire path of the operation as fully asynchronous from the top down. Any synchronous I/O is just a trivial wrapper for performing asynchronous I/O and waiting for its completion [2].
  - Synchronous and asynchronous paths can be separate, to an extent, and can be tuned for different performance characteristics (for example, minimal latency versus maximal throughput) [10].
- Approaches for providing asynchrony:
  - Offload the entire I/O to thread pools (these may be either user-level threads, as in glibc, or kernel worker threads).
  - Use a hybrid approach where initiation of I/O occurs asynchronously and notification of completion occurs synchronously using a pool of waiting threads ([3] and [13]).
  - Implement a true asynchronous state machine for every operation [10].
- Mechanisms for handling user-context dependencies:
  - Convert buffers or other such state into a context-independent form at I/O submission (e.g., by mapping down user-pages) [10].
  - Maintain dedicated per-address space service threads to execute context-dependent steps in the caller's context [3].

The internal design of the AIO support available for Linux 2.4 and the support integrated into Linux 2.5 differ on the above key features. Those differences will be discussed in some detail in later sections.

Other design aspects and issues that are relevant to AIO but which are outside the main focus of this paper include ([8] describes some of these issues in detail):

- The sequencing of operations and steps within an operation that supports

POSIX_SYNCHRONIZED_IO and **POSIX_PRIORITIZED_IO** requirements ([5]), as well as the extent of flexibility to order or parallelize requests to maximize throughput within reasonable latency bounds.

- AIO throttling: deciding on the queue depths and returning an error (**-EAGAIN**) when the depth is exceeded, or if resources are unavailable to complete the request (rather than forcing the process to sleep).
- Completion notification, queuing, and wakeup policies including the design of a flexible completion notification API and optimization considerations like cache-warmth, latency and batching. In the Linux AIO implementation, every AIO request is associated with a completion queue. One or more application threads explicitly wait on this completion queue for completion event(s), where flexible grouping is determined at the time of I/O submission. An exclusive LIFO wakeup policy is used among multiple such threads, and a wakeup is issued whenever I/O completes.
- Support for I/O cancellation.
- User/kernel interface compatibility (per POSIX).

2.3 2.4 Design

The key characteristics of the AIO implementation for the Linux 2.4 kernel are described in [8] and available as patches at [10]. The design:

- Implements asynchronous I/O paths and interfaces separately, leaving existing synchronous I/O paths unchanged. Reuse of existing, underlying asynchronous code is done where possible, for example, raw I/O.
- Implements an asynchronous state machine for all operations. Processing occurs in a series of non-blocking steps driven by asynchronous waitqueue callbacks. Each stage of processing completes with the queueing of deferred work using "work-to-do" primitives. Sufficient state is saved to proceed with the next step, which is run in the context of a kernel thread.
- Maps down user pages during I/O submission. Modifies the logic to transfer data to/from mapped user pages in order to remove the user-context dependency for the copy to/from userspace buffers.

The advantages of these choices are:

- Synchronous I/O performance is unaffected by asynchronous I/O logic, which allows AIO to be implemented and tuned in a way that is optimum for asynchronous I/O patterns.
- The work-to-do primitive permits state to be carried forward to enable continuation from exactly where processing left off when a blocking point was encountered.
- The need for additional threads to complete the I/O transfer in the caller's context is avoided.

There are, however, some disadvantages to the AIO implementation (patchset) for Linux 2.4:

- The duplication of logic between synchronous and asynchronous paths makes the code difficult to maintain.

- The asynchronous state machine is a complex model and therefore more prone to errors and races that can be hard to debug.
- The implementation can lead to inefficient utilization of TLB mappings, especially for small buffers. It also forces the pinning down of all pages involved in the entire I/O request.

These problems motivated a new approach for the implementation of AIO in the Linux 2.5 kernel.

2.4 2.5 Design

Although the AIO design for Linux 2.5 uses most of the core infrastructure from the 2.4 design, the 2.5 design is built on a very different model:

- Asynchronous I/O has been made a first-class citizen of the kernel. Now AIO paths underlie regular synchronous I/O interfaces instead of just being grafted from the outside.
- A retry-based model replaces the earlier work-to-do state-machine implementation. Retries are triggered through asynchronous notification as each step in the process completes. However in some cases, such as direct I/O, asynchronous completion notification occurs directly from interrupt context without requiring any retries.
- User-context dependencies are handled by making worker threads take on (i.e., temporarily switch to) the caller's address space when executing retries.

In a retry-based model, an operation executes by running through a series of iterations. Each iteration makes as much progress as possible in a non-blocking manner and returns. The model assumes that a restart of the operation from where it left off will occur at the next opportunity. To ensure that another opportunity indeed arises, each iteration initiates steps towards progress without waiting. The iteration then sets up to be notified when enough progress has been made and it is worth trying the next iteration. This cycle is repeated until the entire operation is finished.

The implications and issues associated with the retry-based model are:

- Tuning for the needs of both synchronous and asynchronous I/O can be difficult because of the issues of latency versus throughput. Performance studies are needed to understand whether AIO overhead causes a degradation in synchronous I/O performance. It is expected that the characteristics are better when the underlying operation is already inherently asynchronous or rewritten to an asynchronous form, rather than just modified in order to be retried.
- Retries pass through some initial processing steps each time. These processing steps involve overhead. Saving state across retries can help reduce some of the redundant regeneration, albeit with some loss of generality.
- Switching address spaces in the retry thread can be costly. The impact would probably be experienced to a greater extent when multiple AIO processes are running. Performance studies are needed to determine if this is a problem.

Note that I/O cancellation is easier to handle in a retry-based model; any future retries can simply be disabled if the I/O has been cancelled.

Retries are driven by AIO workqueues. If a retry does not complete in a very short time, it can delay other AIO operations that are underway in the system. Therefore, tuning the AIO workqueues and the degree of asynchrony of retry instances each have a bearing on overall system performance.

3 AIO support for filesystem I/O

The Linux VFS implementation, especially as of the 2.5 kernel, is well-structured for retry-based I/O. The VFS is already capable of processing and continuing some parts of an I/O operation outside the user's context (e.g., for readahead, deferred writebacks, syncing of file data and delayed block allocation). The implementation even maintains certain state in the `inode` or address space to enable deferred background processing of writeouts. This ability to maintain state makes the retry model a natural choice for implementing filesystem AIO.

Linux 2.5 is currently without real support for regular (buffered) filesystem AIO. While ext2, JFS and NFS define their `aio_read` and `aio_write` methods to default to `generic_file_aio_read/write`, these routines show fully synchronous behavior unless the file is opened with **O_DIRECT**. This means that an `io_submit` can block for regular AIO read/write operations while the application assumes it is doing asynchronous I/O.

Our implementation of the retry model for filesystem AIO, available as a patchset from [6], involved identifying and focusing on the most significant blocking points in an operation. This was followed by observations from initial experimentation and profiling results, and the conversion of those blocking points to retry exit points.

The implementation we chose starts retries at a very high level. Retries are driven directly by the AIO infrastructure and kicked off via asynchronous waitqueue functions. In synchronous I/O context, the default waitqueue entries are synchronous and therefore do not cause an exit at a retry point.

One of the goals of our implementation for filesystem AIO was to minimize changes to existing synchronous I/O paths. The intent was to achieve a reasonable level of asynchrony in a way that could then be further optimized and tuned for workloads of relevance.

3.1 Design decisions

- Level at which retries are triggered:

  The high-level AIO code retries filesystem read/write operations, passing in the remaining parts of the buffer to be read or written with each retry.

- How and when a retry is triggered:

  Asynchronous waitqueue functions are used instead of blocking waits to trigger a retry (to "kick" a dormant `iocb` into action) when the operation is ready to continue.

  Synchronous routines such as `lock_page`, `wait_on_page_bit`, and `wait_on_buffer` have been modified to asynchronous variations. Instead of blocking, these routines queue an asynchronous wait and return with a special return code, **-EIOCBRETRY**.

  The return value is propagated all the way up to the invoking AIO handler. For this process to work correctly, the calling routine at each level in the call chain needs to break out gracefully if a callee returns the **-EIOCBRETRY** exit code.

- Operation-specific state preserved across retries:

In our implementation [7], the high-level AIO code adjusts the parameters to read or write as retries progress. The parameters are adjusted by the retry routine based on the return value from the filesystem operation indicating the number of bytes transferred.

A recent patch by Benjamin LaHaise [11] proposes moving the filesystem API `read/write` parameter values to the `iocb` structure. This change would enable retries to be triggered at the API level rather than through a high-level AIO handler.

- Extent of asynchrony:

  Ideally, an AIO operation is completely non-blocking. If too few resources exist for an AIO operation to be completely non-blocking, the operation is expected to return **-EAGAIN** to the application rather than cause the process to sleep while waiting for resources to become available.

  However, converting all potential blocking points that could be encountered in existing file I/O paths to an asynchronous form involves trade-offs in terms of complexity and/or invasiveness. In some cases, this tradeoff produces only marginal gains in the degree of asynchrony.

  This issue motivated a focus on first identifying and tackling the major blocking points and less deeply nested cases to achieve maximum asynchrony benefits with reasonably limited changes. The solution can then be incrementally improved to attain greater asynchrony.

- Handling synchronous operations:

  No retries currently occur in the synchronous case. The low-level code distinguishes between synchronous and asynchronous waits, so a break-out and retry occurs only in the latter case while the process blocks as before in the event of a synchronous wait. Further investigation is required to determine if the retry model can be used uniformly, even for the synchronous case, without performance degradation or significant code changes.

- Compatibility with existing code:

  – Wrapper routines are needed for synchronous versions of asynchronous routines.

  – Callers that cannot handle asynchronous returns need special care e.g., making sure that a synchronous context is specified to potentially asynchronous callees.

  – Code that can be triggered in both synchronous and asynchronous mode may present some tricky issues.

  – Special cases like code that may be called via page faults in asynchronous context may need to be treated carefully.

3.2 Filesystem AIO read

A filesystem read operation results in a page cache lookup for each full or partial page of data requested to be read. If the page is already in the page cache, the read operation locks the page and copies the contents into the corresponding section of the user-space buffer. If the page isn't cached, then the read operation creates a new page-cache page and issues I/O to read it in. It may, in fact, read ahead several pages at the same time. The read operation then waits for the I/O to complete (by waiting for a lock on the page), and then performs the copy into user space.

Based on initial profiling results the crucial blocking points identified in this sequence were found to occur in:

- `lock_page`
- `cond_resched`
- `wait_on_page_bit`

Of these routines the following were converted to retry exit points by introducing corresponding versions of the routines that accept a wait-queue entry parameter:

```
lock_page --> lock_page_wq
wait_on_page_bit -->
   wait_on_page_bit_wq
```

When a blocking condition arises, these routines propagate a return value of **-EIOCBRETRY** from `generic_file_aio_read`. When unblocked, the waitqueue routine which was notified activates a retry of the entire sequence.

As an aside, the existing readahead logic helps reduce retries for AIO just as it helps reduce context switches for synchronous I/O. In practice, this logic does not actually cause a volley of retries for every page of a large sequential read.

The following routines are other potential blocking points that may occur in a filesystem read path that have not yet been converted to retry exits:

- `cond_resched`
- meta-data read (get block and read of block bitmap)
- request-queue congestion
- atime updates (corresponding journal updates)

Making the underlying readpages operation asynchronous by addressing the last three blocking points above might require more detailed work. Initial results indicate that significant gains have already been realized without doing so.

3.3 Filesystem AIO write

The degree of blocking involved in a synchronous write operation is expected to be less than in the read case. This is because (unless **O_SYNC** or **O_DSYNC** are specified) a write operation only needs to wait until file blocks are mapped to disk and data is copied into (written to) the page cache. The actual write out to disk typically happens in a deferred way in the context of background kernel threads or earlier in the process via an explicit sync operation. However, for throttling reasons, a wait for pending I/O may also occur in write context.

Some of the more prominent blocking points identified in the this sequence were found to occur in:

- `cond_resched`
- `wait_on_buffer` (during a get block operation)
- `find_lock_page`
- `blk_congestion_wait`

Of these, the following routines were converted to retry exit points by introducing corresponding versions of the routines that accept a wait-queue entry parameter:

```
down --> down_wq
wait_on_buffer --> wait_on_buffer_wq
sb_bread  --> sb_bread_wq
ext2_get_block --> ext2_get_block_wq
find_lock_page --> find_lock_page_wq
blk_congestion_wait -->
     blk_congestion_wait_wq
```

The asynchronous get block support has currently been implemented only for ext2, and only used by `ext2_prepare_write`. All other instances where a filesystem-specific get block routine is involved use the synchronous version. In view of the kind of I/O patterns expected for AIO writes (for example, database workloads), block allocation has not been a focus for conversion to asynchronous mode.

The following routines are other potential blocking points that could occur in a filesystem write path that have not yet been converted to retry exits:

- `cond_resched`
- other meta-data updates, journal writes

Also, the case where **O_SYNC** or **O_DSYNC** were specified at the time when the file was opened has not yet been converted to be asynchronous.

3.4 Preliminary observations

Preliminary testing to explore the viability of the above-described approach to filesystem AIO support reveals a significant reduction in the time spent in `io_submit` (especially for large reads) when the file is not already cached (for example, on first-time access). In the write case, asynchronous get block support had to be incorporated to obtain a measurable benefit. For the cached case, no observable differences were noted, as expected. The patch does not appear to have any effect on synchronous read/write performance.

A second experiment involved temporarily moving the retries into the `io_getevents` context rather than into worker threads. This move enabled a sanity check using `strace` to detect any gross impact on CPU utilization.

Thorough performance testing is underway to determine the effect on overall system performance and to identify opportunities for tuning.

3.5 Issues and todos

- Should the `cond_resched` calls in read/write be converted to retry points?
- Are asynchronous get block implementations needed for other filesystems (e.g., JFS)?
- Optional: should the retry model be used for direct I/O (DIO) or should synchronous DIO support be changed to wait for the completion of asynchronous DIO?
- Should relevant filesystem APIs be modified to add an explicit waitqueue parameter?
- Should the `iocb` state be updated directly by the filesystem APIs or by the high-level AIO handler after every retry?

4 AIO support for direct I/O

Direct I/O (raw and **O_DIRECT**) transfers data between a user buffer and a device without copying the data through the kernel's buffer cache. This mechanism can boost performance if the data is unlikely to be used again in the short term (during a disk backup, for example), or for applications such as large database management systems that perform their own caching.

Direct I/O (DIO) support was consolidated and redesigned in Linux 2.5. The old scalability problems caused by preallocating kiobufs and buffer heads were eliminated by virtue of the new BIO structure. Also, the 2.5 DIO code streams the entire I/O request (based on the underlying driver capability) rather than breaking the request into sector-sized chunks.

Any filesystem can make use of the DIO support in Linux 2.5 by defining a `direct_IO` method in the `address_space_operations` structure. The method must pass back to the DIO code a filesystem-specific get block function, but the DIO support takes care of everything else.

Asynchronous I/O support for DIO was added in Linux 2.5. The following caveats are worth noting:

- Waiting for I/O is done asynchronously but multiple points in the submission codepath can potentially cause the process to block (such as the pinning of user pages or processing in the filesystem get block routine).
- The DIO code calls `set_page_dirty` before performing I/O since the routine must be called in process context. Once the I/O completes, the DIO code—operating in interrupt context—checks whether the pages are still dirty. If so, nothing further is done; otherwise, the pages are made dirty again via a workqueue run in process context.

5 Vector AIO

5.1 2.5 readv/writev improvements

In Linux 2.5, direct I/O (raw and **O_DIRECT**) readv/writev was changed to submit all segments or iovecs of a request before waiting for I/O completion. Prior to this change, DIO readv/writev was processed in a loop by calling the filesystem read/write operations for each iovec in turn.

The change to submit the I/O for all iovecs before waiting was a critical performance fix for DIO. For example, tests performed on an aic-attached raw disk using 4Kx8 readv/writev showed the following improvement:

Random writev	8.7 times faster
Sequential writev	6.6 times faster
Random readv	sys time improves 5x
Sequential readv	sys time improves 5x
Random mixed I/O	5 times faster
Sequential mixed I/O	6.6 times faster

5.2 AIO readv/writev

With the DIO readv/writev changes integrated into Linux 2.5, we considered extending the functionality to AIO. One problem is that AIO readv/writev ops are not defined in the `file_operations` structure, nor are readv/writev part of the AIO API command set. Further, the interface to `io_submit` is already an array of `iocb` structures analogous to the vector of a readv/writev request, so a real question is whether AIO readv/writev support is even needed. To answer the question, we prototyped the following changes [12]:

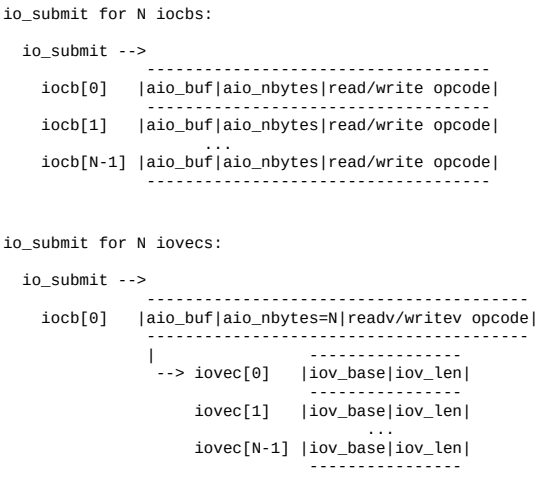
- added `aio_readv/writev` ops to the `file_operations` structure
- defined `aio_readv/writev` ops in the raw driver
- added 32- and 64-bit readv and writev command types to the AIO API
- added support for readv/writev command types to `fs/aio.c`:

```
fs/aio.c                |  156 ++++
include/linux/aio.h     |    1
include/linux/aio_abi.h |   14 ++++
include/linux/fs.h      |    2
4 files changed, 173 insertions(+)
```

5.3 Preliminary results

With the above-noted changes, we were able to test whether an `io_submit` for N iovecs is

more performant than an `io_submit` for N iocbs.



Based on preliminary data [1] using direct I/O, an `io_submit` for N iovecs outperforms an `io_submit` for N iocbs by as much as two-to-one. While there is a single `io_submit` in both cases, aio readv/writev shortens codepath (i.e., one instead of N calls to the underlying driver method) and normally results in fewer bios/callbacks.

5.4 Issues

The problem with the proposed support for AIO readv/writev is that it creates code redundancy in the custom and generic filesystem layers by adding two more methods to the `file_operations` structure. One solution is to first collapse the `read/write/readv/writev/aio_read/aio_write` methods into simply `aio_read` and `aio_write` and to convert those methods into vectored form [11].

6 Performance

6.1 System setup

All benchmarks for this paper were performed on an 8-way 700MHz Pentium™III machine with 4GB of main memory and a 2MB L2 cache. The disk subsystem used for the I/O tests consisted of 4 IBM® ServeRAID-4H™dual-channel SCSI controllers with 10 9GB disk drives per channel totalling 80 physical drives. The drives were configured in sets of 4 (2 drives from each channel) in a RAID-0 configuration to produce 20 36GB logical drives. The software on the system was SuSE Linux Enterprise Server 8.0. Where noted in the results, the kernel was changed to 2.5.68 plus required patches [7]. For AIO benchmarking, libaio-0.3.92 was installed on the system.

6.2 Microbenchmark

The benchmark program used to analyze AIO performance is a custom benchmark called rawiobench [9]. Rawiobench spawns multiple threads to perform I/O to raw or block devices. It can use a variety of APIs to perform I/O including `read/write`, `readv/writev`, `pread/pwrite` and `io_submit`. Support exists for both random and sequential I/O and the exact nature of the I/O request is dependent on the actual test being performed. Each thread runs until all threads have completed a minimum number of I/O operations at which time all of the threads are stopped and the total throughput for all threads is calculated. Statistics on CPU utilization are tracked during the run.

The rawiobench benchmark will be run a number of different ways to try to characterize the performance of AIO compared to synchronous I/O. The focus will be on the 2.5 kernel.

The first comparison is designed to measure the overhead of the AIO APIs versus using the normal read/write APIs (referred to as the "overhead" test). For this test rawiobench will be run using 160 threads each doing I/O to one of the 20 logical drives for both sequential and random cases. In the AIO case, an

`io_submit/io_get_events` pair is submitted in place of the normal read or write call. The baseline synchronous tests will be referred to in the charts simply as "seqread" "seqwrite" "ranread" "ranwrite" with an extension of either "ODIR" for a block device opened with the **O_DIRECT** flag or "RAW" for a raw device. For the AIO version of this test, "aio" is prepended to the test name (e.g., aioseqread).

The second comparison is an attempt to reduce the number of threads used by AIO and to take advantage of the ability to submit multiple I/Os in a single request. To accomplish this the number of threads for AIO was reduced from 8 per device to 8 total (down from 160). Each thread is now responsible for doing I/O to every device instead of just one. This is done by building an `io_submit` with 20 I/O requests (1 for each device). The process waits for all I/Os to complete before sending new I/Os. This AIO test variation is called "batch mode" and is referred to in the charts by adding a "b" to the front of the test name (e.g., bseqaioread).

The third comparison will improve upon the second by having each AIO process calling `io_getevents` with a minimum number equal to 1 so that as soon as any previously submitted I/O completes, a new I/O will be driven. This AIO test variation is called "minimum batch mode" and is referred to in the charts by adding a "minb" to the front to the test name (e.g., minbseqaioread).

In all sequential test variations, a global offset variable per device is used to make sure that each block is read only once. This offset variable is modified using the `lock xadd` assembly instruction to ensure correct SMP operation.

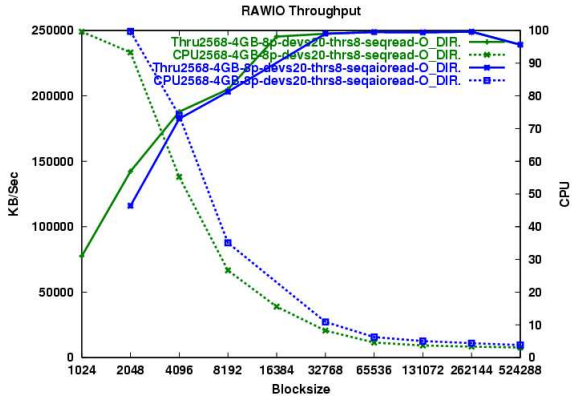


Figure 1: Sequential Read Overhead

6.3 Results, comparison and analysis

Raw and **O_DIRECT** performed nearly identically on all of the benchmarks tests. In order to reduce redundant data and analysis, only the data from **O_DIRECT** will be presented here.

For the first comparison of AIO overhead, the results show that there is significant overhead to the AIO model for sequential reads (Figure 1). For small block sizes where the benchmark is CPU bound, the AIO version has significantly lower throughput values. Once the block size reaches 8K and we are no longer CPU bound, AIO catches up in terms of throughput, but averages about 20% to 25% higher CPU utilization.

In Figure 2 we can see the performance of random reads using AIO is identical to synchronous I/O in terms of throughput, but averages approximately 20% higher CPU utilization. By comparing synchronous random read plus seeks with random pread calls (Figure 3) we see that there is minimal measurable overhead associated with having two system calls instead of one. From this we can infer that the overhead seen using AIO in this test is associated with the AIO internals, and not the cost of the additional API call. This overhead seems

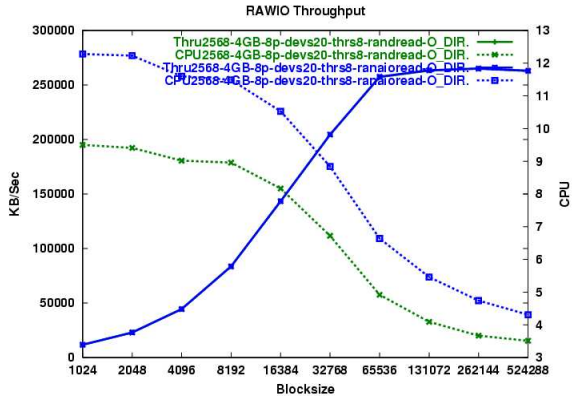


Figure 2: Random Read Overhead

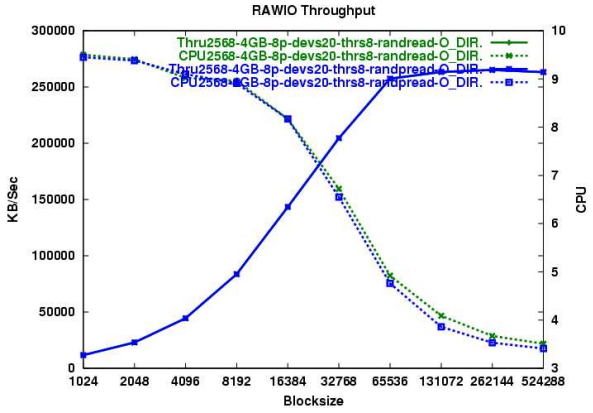


Figure 3: read vs. pread

excessive and probably indicates a problem in the AIO kernel code. More investigation is required to understand where this extra time is being spent in AIO.

For write performance we can see that AIO achieves approximately the same level of throughput as synchronous I/O, but at a significantly higher cost in terms of CPU utilization at smaller block sizes. For example, during the sequential write test at 2K block sizes, AIO uses 97% CPU while synchronous uses only 55%. This mirrors the behavior we see in the read tests and is another indication of problems within the AIO code.

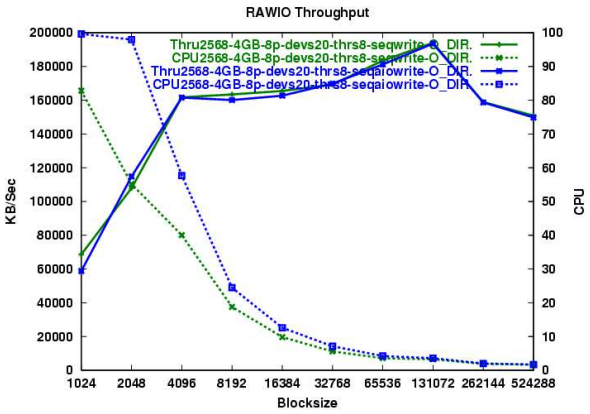


Figure 4: Sequential Write Overhead

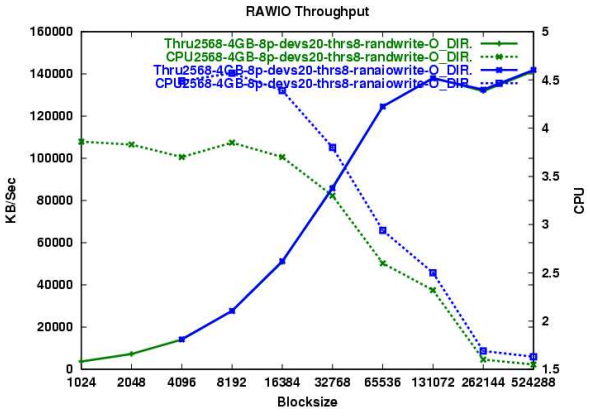


Figure 5: Random Write Overhead

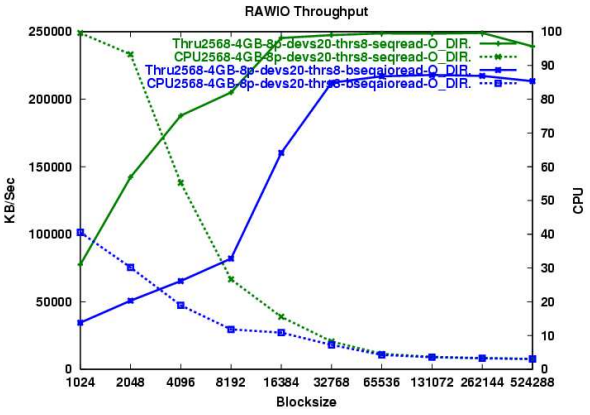


Figure 6: Sequential Read Batch

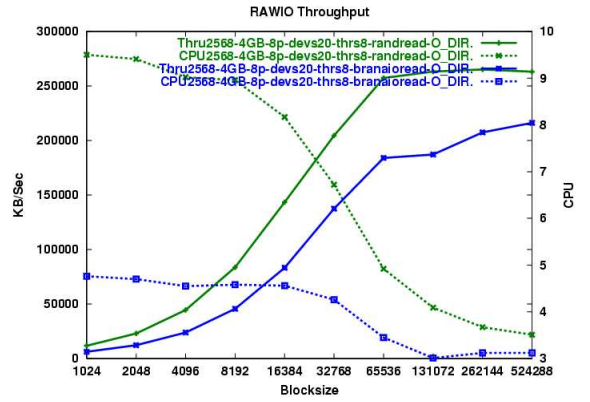


Figure 7: Random Read Batch

Batch mode AIO performs poorly on the sequential and random read tests. Throughput is significantly lower as is CPU utilization (Figures 6,7). This is probably due to the fact that we can drive more I/Os in a single request than the I/O subsystem can handle, but we must wait for all the I/Os to complete before continuing. This results in multiple drives being idle while waiting for the last I/O in a submission to complete.

AIO batch mode also under-performs on the write tests. While CPU utilization is lower, it never achieves the same throughput values as synchronous I/O. This can be seen in Figures 8 and 9.

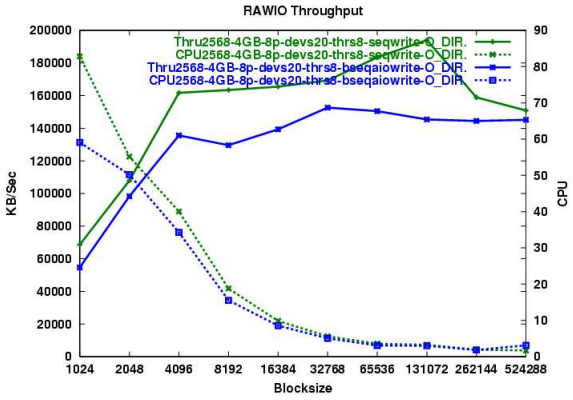


Figure 8: Sequential Write Batch

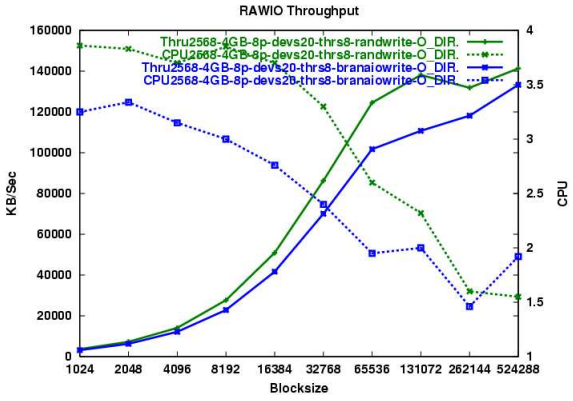


Figure 9: Random Write Batch

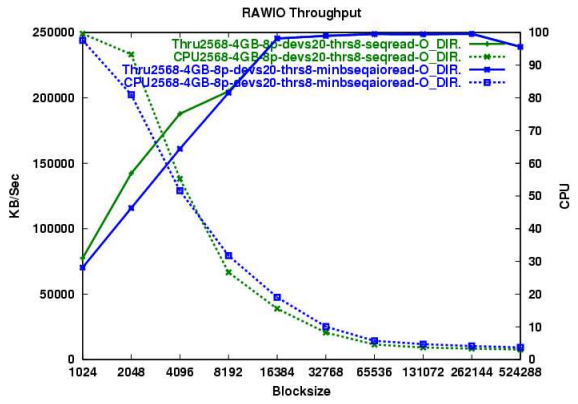


Figure 10: Sequential Read Minimum Batch

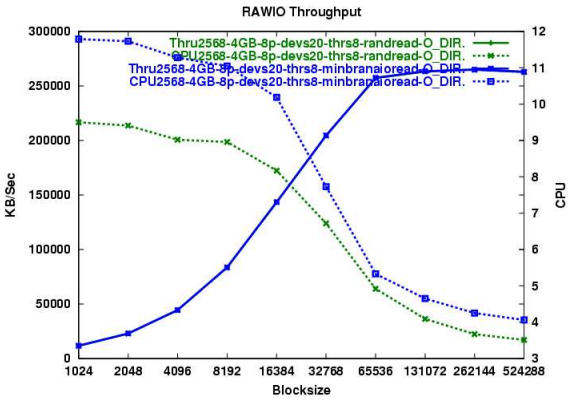


Figure 11: Random Read Minimum Batch

Minimum batch mode improves considerably on the overhead and batch mode tests; however, for **O_DIRECT** access AIO either lags in throughput or uses more CPU for all block sizes in the sequential read test (Figure 10). For the random read test minimum batch mode AIO has identical throughput to synchronous reads, but uses from 10% to 20% more CPU in all cases (Figure 11). Sequential minimum batch mode AIO comes close to the performance (both throughput and CPU utilization) of synchronous, but does not ever perform better.

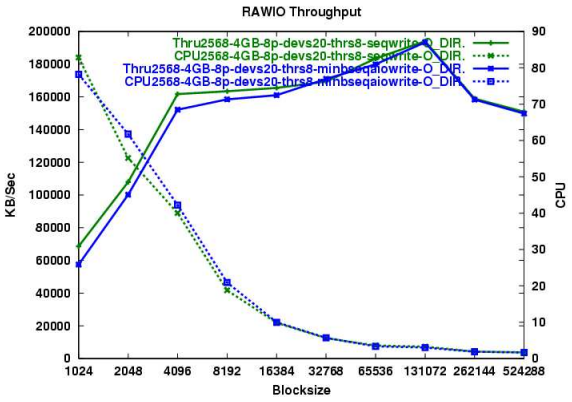


Figure 12: Sequential Write Minimum Batch

Minimum batch mode sequential writes, like reads, lag behind synchronous writes in terms of overall performance (Figure 12). This difference gets smaller and smaller as the block size increases. For random writes (Figure 9), the difference increases as the block size increases.

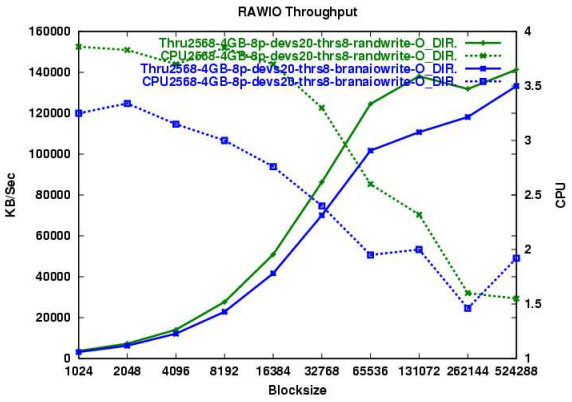


Figure 13: Random Write Minimum Batch

Since we are seeing lower CPU utilization for minimum batch mode AIO at smaller block sizes we tried increasing the number of threads in that mode to see if we could drive higher I/O throughput. The results seen in Figure 14 show that indeed for smaller block sizes that using 16 threads instead of 8 did increase the throughput, even beating synchronous I/O at the 8K block size. For block sizes larger than 8K the

increase in number of threads either made no difference, or causes a degradation in throughput.

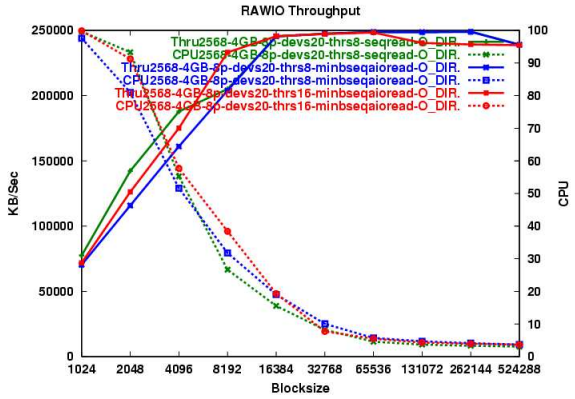


Figure 14: Sequential Read Minimum Batch with 16 threads

In conclusion, there appears to be no conditions for raw or **O_DIRECT** access under which AIO can show a noticeable benefit. There are however, cases where AIO will cause reductions in throughput and higher CPU utilization. Further investigation is required to determine if changes in the kernel code can be made to improve the performance of AIO to the level of synchronous I/O.

It should be noted that at the larger block sizes, CPU utilization is so low (less than 5%) for both synchronous I/O and AIO that the difference should not be an issue. Since using minimum batch mode achieves nearly the same throughput as synchronous I/O for these large block sizes, an application could choose to use AIO without any noticeable penalty. There may be cases where the semantics of the AIO calls make it easier for an application to coordinate I/O, thus improving the overall efficiency of the application.

6.3.1 Future work

The patches which are available to enable AIO for buffered filesystem access are not stable enough to collect performance data at present. Also, due to time constraints, no rawiobench testcases were developed to verify the effectiveness of the readv/writev enhancements for AIO [12]. Both items are left as follow-on work.

7 Acknowledgments

We would like to thank the many people on the `linux-aio@kvack.org` and `linux-kernel@vger.kernel.org` mailing lists who provided us with valuable comments and suggestions during the development of these patches. In particular, we would like to thank Benjamin LaHaise, author of the Linux kernel AIO subsystem. The retry based model for AIO, which we used in the filesystem AIO patches, was originally suggested by Ben.

This work was developed as part of the Linux Scalability Effort (LSE) on SourceForge (`sourceforge.net/projects/lse`). The patches developed by the authors and mentioned in this paper can be found in the "I/O Scalability" package at the LSE site.

8 Trademarks

References

[1] AIO readv/writev performance data. `http://osdn.dl.sourceforge.net/sourceforge/lse/vector-aio.data`.

[2] Asynchronous I/O on Windows® NT.

[3] Kernel Asynchronous I/O implementation for Linux from SGI. `http://oss.sgi.com/projects/kaio`.

[4] POSIX Asynchronous I/O.

[5] Open Group Base Specifications Issue 6 IEEE Std 1003.1. `http://www.opengroup.org/onlinepubs/007904975/toc.htm`, 2003.

[6] Suparna Bhattacharya. 2.5 Linux Kernel Asynchronous I/O patches. `http://sourceforge.net/projects/lse`.

[7] Suparna Bhattacharya. 2.5 Linux Kernel Asynchronous I/O rollup patch. `http://osdn.dl.sourceforge.net/sourceforge/lse/aiordwr-rollup.patch`.

[8] Suparna Bhattacharya. Design Notes on Asynchronous I/O for Linux. `http://lse.sourceforge.net/io/aionotes.txt`.

[9] Steven Pratt Bill Hartner. rawiobench microbenchmark. `http://www-124.ibm.com/developerworks/opensource/linuxperf/rawread/rawr%ead.html`.

[10] Benjamin LaHaise. 2.4 Linux Kernel Asynchronous I/O patches. `http://www.kernel.org/pub/linux/kernel/people/bcrl/aio/patches/`.

[11] Benjamin LaHaise. Collapsed read/write iocb argument-based filesystem interfaces. `http://marc.theaimsgroup.com/?l=linux-aio&m=104922878126300\&w=2`.

[12] Janet Morgan. 2.5 Linux Kernel Asynchronous I/O readv/writev patches. `http://marc.theaimsgroup.com/?l=linux-aio&m=103485397403768\&w=2`.

[13] Venkateshwaran Venkataramani Muthian Sivathanu and Remzi H. Arapaci-Dusseau. Block Asynchronous I/O: A Flexible Infrastructure for User Level Filesystems. `http://www.cs.wisc.edu/~muthian/baio-paper.pdf`.